CONTENTS

INTRODUCTION

This manual is used in conjunction with the Atari Assembler Cartridge. To use the Assembler Cartridge and this manual you should already have the background described below.

1. BACKGROUND YOU NEED

A. You should be familiar with the general capabilities of the Atari 400/800 personal computer and familiarity with its keyboard and all screen-editing functions. This material is covered in the Operators' Manual supplied with the 400 and the 800.

B. You should have a working knowledge of 650x assembly language, including knowledge of the effect of every instruction. This may be obtained from several sources. There is a high degree of standardization of 650x Assembly Language mnemonics. There is more variation in ways of writing operands. A complete summary of instructions is given in Appendix A. Some descriptions in this manual take the form of a reminder or brief review and assume that the reader is already familiar with the subject.

Shouldn't it be included

C. Finally, if you are using the Assembler Cartridge in a system with diskette drive, you should have practical knowledge of the Atari Diskette Operating System (DOS).

1.b.1
1.a.1

Most users of the Atari 400/800 computers will program in Atari BASIC, the standard high-level interpretive language supplied with the computer. BASIC translates instructions into machine code a line at a time. BASIC is inefficient in two senses: it does not execute quickly and it is "wasteful" of memory. If you need to have a process caried out more efficiently than can be achieved in BASIC, then you should use Assembly Language.

In many applications it may be satisfactory to write most of your program in BASIC and write critical parts of it in Assembly Language. The Atari BASIC manual gives an example of this facility, with the USR Function of BASIC.

You should read this manual straight through. You will need most of the information presented. The manual is organized so that it may serve as a convenient reference book after you have acquired facility with the Assembler.

2. THE ASSEMBLER CARTRIDGE

The Assembler Cartridge contains three separate programs, called the WRITER/EDITOR, ASSEMBLER and DEBUGGER.

You write Assembly Language programs with the help of the WRITER/EDITOR. You then assemble your program (translate into machine language) with the ASSEMBLER. To modify the program you use the WRITER/EDITOR or the DEBUGGER. To test or trace the actual operation of your program you use the DEBUGGER.

one using source other object

not clear

1.B.2
1.a.2 10/26

3. WHAT YOU OUGHT TO HAVE TO USE THE ASSEMBLER CARTRIDGE

To take advantage of the Assembler Cartridge, it is essential to posses only the 400 or the 800 and the Assembler Cartridge itself. However, without a permanent storage device you will have to enter your program on the keyboard each time that you wish to use it. This can be so tedious that a cassette recorder or a diskette drive is a practical necessity.

poor

The 410 cassette recorder is supplied with the 800 system, but we do not recommend its use with the Assembler Cartridge. We recommend that a diskette drive, the Atari 820, be used to provide permanent storage of programs.

A diskette drive requires a Diskette Operating System, which needs at least 16k of memory. Consequently, your Atari 400 cannot use a diskette drive unless you upgrade your computer from 18k to 16k of memory.

The Atari 820 Printer is an optional addition to your system. It will allow you to keep a record of your programs in a form that you can read.

WHAT YOU OUGHT TO HAVE TO USE THE ASSEMBLER CARTRIDGE

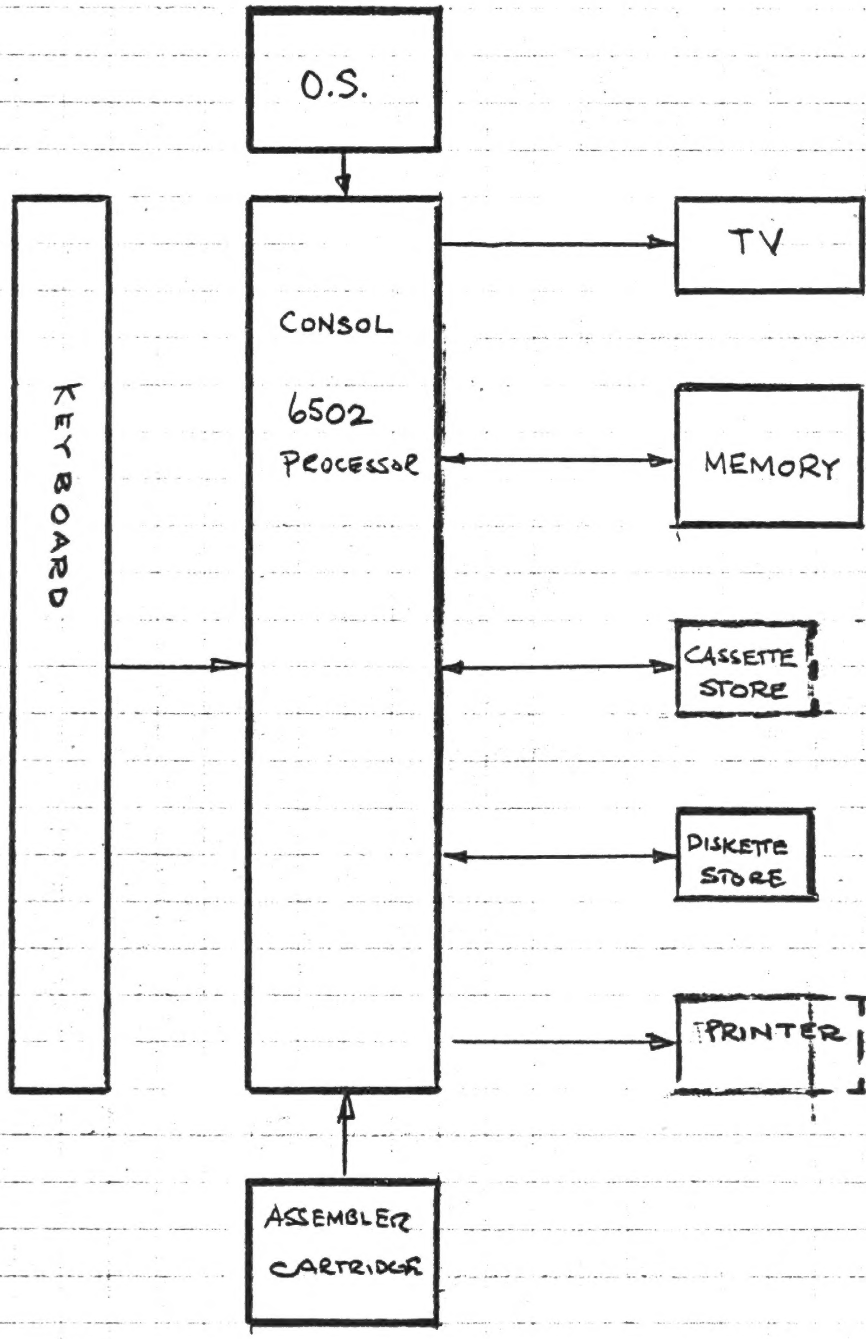


Figure 1. The elements of the system recommended for using the Assembler Cartridge.
(Broken-line elements are optional)

5.1 The 6502 Micro Processor

The 6502 micro processor is represented in Figure 2

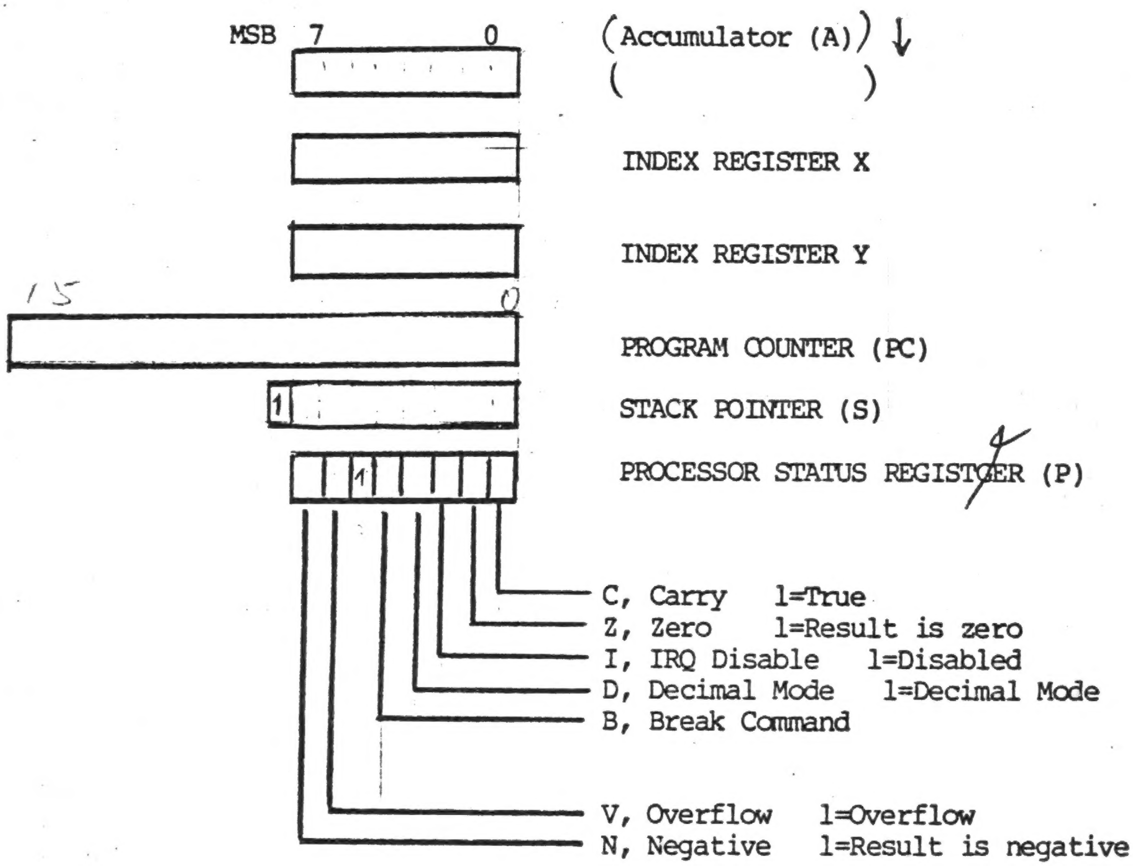


Fig 2

Figure 2. The 6502 Microprocessor from the Programmer's Point of View.

The A register is referred to as the Accumulator. The Registers X and Y are Index Registers used as scratch pads and in certain addressing modes, as explained later (See Addressing Modes) The Program Counter is the counter incremented as each instruction is fetched and executed. The numbers it contains correspond to the locations in memory of successive instructions.

The Processor Status Register is updated after execution of each instruction, according to the effects of the instruction. Memory consists of 64k possible locations addressable by the Program Counter.

Not all of these 64k locations are available in an actual system. That is because some addresses are in RAM, and some are addresses of devices in the computer. The allocation of addresses is given in Appendix B.

The Stack occupies memory from 01FF down to 0100. Four special instructions access the stack. The stack pointer register always points to the top of the stack.

You can specify or determine the contents of all registers and all memory locations by Assembly Language instructions. It is assumed in this manual that you already know how to manipulate numbers in this manner.

l.

5.2 PROCESSOR STATUS FLAGS

Seven specific effects of an instruction are stored in the Processor Status Register (Register P) as each instruction is executed. The bit position of the effect is shown in Figure 2. Each bit is also called a flag. The presence or absence of an effect is shown by whether the flag is set to 1 or 0. The flags are called CARRY, ZERO, INTERRUPT, DECIMAL, BREAK, OVERFLOW and NEGATIVE, abbreviated C, Z, I, D, B, V and N.

There are two instructions that affect the stack and also determine the content of Register P. These two instructions are PLP (PULL REGISTER P) and RTI (RETURN FROM INTERRUPT). In the course of execution of each of these instructions a byte is pulled from the stack on to Register P. The usual purpose is to restore Register P to some previously saved value. It is understood that the two instructions may change the value of every condition flag.

l.c

C FLAG (CARRY)

The Carry flag is modified when certain arithmetic operations generate a carry. The flag can also be set and cleared by direct instructions. In the case of shift and rotate instructions, the carry bit is used as a ninth bit in a manner similar to the carry of arithmetic operations.

Instructions to set and clear the Carry flag are

SEC SET CARRY
CLC CLEAR CARRY

Other instructions whose execution may affect the Carry are

ADC ADD WITH CARRY
SBC SUBTRACT WITH BORROW
CPX COMPARE WITH X
CPY COMPARE WITH Y
CMP COMPARE
ASL SHIFT LEFT
LSR SHIFT RIGHT
ROL ROTATE LEFT
ROR ROTATE RIGHT

1.b.15

D FLAG (DECIMAL MODE)

The Decimal Mode flag controls whether the adder operates ~~for~~ add and subtract instructions as a bit-wise binary adder or as a decimal adder.

The instructions to control the flag are

SED SET DECIMAL MODE
CLD CLEAR DECIMAL MODE

P.3

B FLAG (BREAK)

The Break flag is set and cleared only by the processor. It is used during an interrupt-service routine to determine whether the interrupt was caused by an interrupt signal or by a BREAK instruction.

I FLAG (INTERRUPT DISABLE)

The Interrupt Disable flag is set and cleared by the processor in the course of controlling operations that require that the processor not be interrupted. There are also instructions to set and clear the flag directly. While the flag is set to 1 the processor ignores interrupt requests.

The Interrupt Disable flag is affected by the hardware devices in the system, as part of the process of controlling communication between elements of the system. The I flag is also affected by the instructions PLP and RTI.

Instructions to set and clear the Interrupt Disable flag are

SEI SET INTERRUPT DISABLE
CLI CLEAR INTERRUPT DISABLE

V FLAG (OVERFLOW)

If you require sign information in the result of an arithmetic operation, you use bit 7. In order to be able to interpret correctly bit 7 in the result of an arithmetic operation you need to know whether two addends with bit 7=0 gave a result with bit 7=1, or *vice versa*. In these cases the V flag is set. The V flag being set indicates that you must use a sign correction routine.

The V flag is also affected by the BIT instruction. The BIT instruction, which may be used to test unknown byte patterns, allows the V flag to show the value of bit 6 in the tested byte.

When used in this way the V flag gives no arithmetic information; it is simply the value of bit 6 of the operand of the BIT instruction.

There is no instruction to set the V flag. The instruction to clear the flag is

CLC CLEAR V FLAG

Other instructions which may affect the V flag are

ADC ADD WITH CARRY
SBC SUBTRACT WITH BORROW
BIT BIT TEST

Z FLAG (ZERO)

The Zero flag is set by the processor during any data movement or calculation when the 8 bits of the result of the operation are zero. Therefore, the flag is set (1) when the result is 0 and cleared (0) when the result is not 0. This feature of the processor permits a simple test for completion when iterating a specific number of times. There are no instructions that directly set or clear the Z flag.

Instructions whose execution may affect the Z flag are

LDA	LOAD ACCUMULATOR
LDX	LOAD X
LDY	LOAD Y
ADC	ADD WITH CARRY
AND	AND WITH ACCUMULATOR
INX	INCREMENT X
INY	INCREMENT Y
INC	INCREMENT
DEX	DECREMENT X
DEY	DECREMENT Y
DEC	DECREMENT
ORA	OR WITH ACCUMULATOR
EOR	XOR WITH ACCUMULATOR
SBC	SUBTRACT WITH BORROW
CMP	COMPARE WITH ACCUMULATOR
CPX	COMPARE WITH X
CPY	COMPARE WITH Y
ASL	SHIFT LEFT
LSR	SHIFT RIGHT
ROR	ROTATE RIGHT
BIT	BIT TEST
TAX	TRANSFER A TO X
TXA	TRANSFER X TO A
TAY	TRANSFER A TP Y
TYA	TRANSFER Y TO A
TSX	TRANSFER S TO X
PLA	PULL STACK TO ACCUMULATOR

N FLAG (NEGATIVE)

The N flag is set to the value of bit 7 of the number that results ~~in~~ from all data movement and arithmetic operations. One consequence of this, for example, is that after an addition in signed arithmetic you can determine the sign of the result directly rather than devising a way to find the value of bit 7 of the result.

All operations that involve data movement into Register A, X and Y affect the N flag, so it is very widely used as a test condition to determine the completion of a process or the existence of a state, ~~using the Branch instructions~~ in conjunction with the branch instructions BMI and BPL. It is as useful in this fashion as the Z flag. Like the Z flag, the N flag only reflects the results of processor operations and there are no instructions that set or clear the flag directly.

The instructions that may affect the N flag are

LDA	LOAD ACCUMULATOR
LDX	LOAD X
LDY	LOAD Y
ADC	ADD WITH CARRY
AND	AND WITH ACCUMULATOR
INX	INCREMENT X
INY	INCREMENT Y
INC	INCREMENT
DEX	DECREMENT X
DEY	DECREMENT Y
DEC	DECREMENT
ORA	OR WITH ACCUMULATOR
EOR	XOR WITH ACCUMULATOR
SBC	SUBTRACT WITH BORROW
CMP	COMPARE WITH ACCUMULATOR
CPX	COMPARE WITH X
CPY	COMPARE WITH Y
ASL	SHIFT LEFT
LSR	SHIFT RIGHT
ROR	ROTATE RIGHT
BIT	BIT TEST
TAX	TRANSFER A TO X
TXA	TRANSFER X TO A
TAY	TRANSFER A TP Y
TYA	TRANSFER Y TO A
TSX	TRANSFER S TO X
PLA	PULL STACK TO ACCUMULATOR

5.3 ADDRESSING MODES

An instruction, according to a useful and widely-used concept, specifies an operation, together with directions to identify the data subject to the operation—the operand. For example, a particular instruction is

LDA 3333

The operation is loading a number into the Accumulator; the number to be loaded is in address 3333.

In the next example, the operation is still loading a number into the Accumulator, but the number to be loaded is given directly.

LDA #29

There are many possible methods of specifying the data to be operated on. Though there are many common methods, different processors use different methods. The methods are classifiable in various ways, too. Each different class of method is usually referred to as an addressing mode. In some cases, classes referred to as addressing modes are ways distinguishing types of operand, or ways of interpreting an operand.

The address that contains the number that is operated on is called the effective address. In all modes of addressing except IMMEDIATE it is the number in the effective address that is added, subtracted, rotated, etc.

7

We give a list of addressing modes below. This is virtually the same as a list of types of operand. After the list of addressing modes we give a list of ways that different operands are written.

In addresses given by bytes 2 and 3 of an instruction, byte 2 gives the low address and byte 3 gives the high address.

Addresses said the be given by byte 2 of an instruction are presumed to be 000x - that is, in Page Zero.

5.3.1 NON-INDEXED ADDRESSING MODES

IMMEDIATE	The operand is byte 2 of the 2-byte instruction.
PAGE ZERO	The effective address is the address given by byte 2 of the 2-byte instruction. Since the high address is presumed to be zero, the effective address is in Page Zero.
ABSOLUTE	The ffective address is byte 2 (low address) and byte 3 (high address) of the 3-byte instruction.

5.3.2 INDEXED ADDRESSING MODES

ABSOLUTE INDEXED	The effective address is determined as for Absolute Addressing, with the addition of the number in Register X (Absolute, X) or Register Y (Absolute, Y) to the Absolute Address.
INDEXED INDIRECT	[(Indirect, X)] The effective address is the 2-byte number starting at the address given by byte 2 of the instruction plus the number in Register X. (Always Page Zero).
INDIRECT INDEXED	[(Indirect),Y] The effective address is the base address plus the number in Register Y. The base address is the 2-byte number starting at the address given by byte 2 of the instruction. (Always Page Zero).
INDEXED ZERO PAGE	[Z Page, X; Z Page, Y] The effective address is the address given by byte 2 of the instruction plus the number in Register X or Y.

5.3.3 MISCELLANEOUS ADDRESSING MODES

REGISTER	The effective address is Register A (Shift and Rotate instructions) or Register X or Y (Increment and Decrement instructions).
IMPLIED	The effective address is the Register, the stack or a flag, as implied by the operation.
RELATIVE	Byte 2 of the instruction is added to the Program Counter.
INDIRECT	The effective address is the 2-byte address starting at the location given by bytes 2 and 3 of the instruction.
NONE	Machine control operations (HLT, NOP) do not address data.

6. PROGRAM FORMAT—HOW TO WRITE A STATEMENT

A program to be assembled is called a source program. It consists of Statements. Each Statement is terminated with RETURN. A Statement must be 1-106 characters.

The Assembler interprets parts of a Statement in one of five different ways—as a Statement Number, Label, Operation Code Mnemonic (or Directive), Operand or Comment. These occupy different character positions in the Statement, Statement Number first and Comment last. The positions occupied by the different parts of a Statement are called "fields".

STATEMENT NUMBER Every statement must start with a number from 0 to 65,535. We recommend that Statements be numbered 10, 20, 30, etc. The WRITER/EDITOR puts the Statements in numerical order. Numbering by 10s allows you to insert new Statements in later versions of a program. In addition, the WRITER/EDITOR has convenient commands for numbering Statements (See NUM,REN).

LABEL A Label is entered on the left-most position of the screen, after the Statement Number. You must leave exactly one space (not a tab) after the Statement Number. The Label must start with a letter and contain only letters and numbers. It can be as short as two characters and as long as the limitation of Statement length permits (107 less the number of characters in the Statement Number).

You are not forced to have a Label. To go on to the next field, enter another space (or a tab). The Assembler will interpret the entries after a tab as an Operation Code Mnemonic.

OPERATION CODE MNEMONIC The Operation Code (or Op Code) Mnemonic must be one of those given in Appendix A. It must be entered in the field that starts at least two spaces after the Statement Number, or after a Label. An Operation Code Mnemonic in the wrong field will not be identifed as an error in the Write/Edit mode, but will be flagged when you assemble the program (Error 6).

OPERAND The field of the Operand starts at least one space (or a tab) after an Operation Code Mnemonic. Some Operation Codes Mnemonics do not require an Operand. The Assembler will expect an Operand if the Op Code Mnemonic requires one. The

form of the Operand that is appropriate to each Operation Code Mnemonic is given in Appendix A and each different way of writing an Operand is given in the section called HOW TO WRITE OPERANDS.

COMMENT A Comment appears on the listing of a program, but does not figure in the assembled form. but is ignored by the assembler

There are two ways to have the Assembler interpret entries as Comments. One way is to make the entries in the Comment field, which occupies the remainder of the line after the instruction field(s). At least one space must separate the fields. There may not be enough space in the Comment field for the Comment you wish to write there. In that case you can extend your Comment into the next line. If you wish to number that line, you will have to use the second way to signal the Assembler that your entries are part of a Comment. To do so, you enter one space and a semi-colon after the Line Number. The Comment so entered need not be a continuation of the Comment on the previous line. Everything between the initial semi-colon and the RETURN is ignored by the Assembler, but will be printed in the listing of the program.

The Statements below show various examples of Comments correctly positioned in the Statement.

```
210 LABL LDX ABS COMMENT IN COMMENT FIELD
220  TSX COMMENT IN COMMENT FIELD
230  PHP COMMENT IN THIS LINE CONT
240 ;INUED ON THIS LINE
250  LDY #FF
260 ;COMMENT ON ITS OWN LINE
```

7. HOW TO WRITE OPERANDS

This section shows how to write operands. The examples use Statement Number 10 (also called Line Number 10). A Statement Number is used because an instruction entered without a Statement Number is not allowed by the WRITER/EDITOR.

The examples use "BY and and "ABS" as a one-byte and a two-byte number, respectively. This use implies that the program includes definitions of BY and ABS as, for example:

```
100 BY=155
200 ABS=567
```

Please refer to the description of the = Directive for an explaination of the definitions of lines 100 and 200.

Hex Operands

A number is interpreted as a decimal number unless it is preceded by "$", in which case it is interpreted as a hex number.

Examples:

```
10  STA $9325
10  ASL $15
10  CPY ABS
```

Immediate Operands

An immediate operand is an operand that contains the data of the instruction. The pound sign (#) must be present to indicate an immediate operand.

Examples: describe

```
10  LDA #12
10  ORA #$3C
10  CPY #BY
```

— Define BY

Absolute Operands

Absolute operands are evaluated as 16-bit numbers.

Examples:

```
10   LDX $1212
10   CPY 2345
10   DEC 579
10   BIT ABS
```

Absolute Indexed Operands

An Absolute Indexed Operand uses Register X or Y. The operand is written ——,X or——,Y

Examples:

```
10  AND $3C26,X
10  EOR 20955,Y
10  STA ABS,Y
```

Indexed Indirect Operands

An Indexed Indirect instruction uses Register X. The operand is written (—,X)

Examples:

```
10  INC ($99,X)
10  ADC (BY,X)
```

Indirect Indexed Operands

An Indirect Indexed instruction uses Register Y. The operand is written (—),Y

Examples:

```
10  LDA ($2B),Y
10  CMP ($E5),Y
10  ORA (BY),Y
```

Indexed Page Zero

A Zero Page Indexed operand is written —,X or —,Y

Examples:

```
10  INC $34,X
10  STX $AB,Y
10  LDX BY,Y
```

String Operands

Operands or parts of operands enclosed in double quotation marks are translated into the ATASCII numbers of the characters between the quotation marks. The instuction or Directive must be appropriate,

Examples:

```
10 ADDR .BYTE "9+1 =s TEN"
```

Execution of this Directive causes the ATASCII numbers corresponding to "9", "+", etc., to be stored at successive locations starting at ADDR. Note that the ATASCII representation of any character except the quotation mark (") can be stored with the .BYTE Directive having a string operand.

PROGRAMMING

Your intial contact with the Assembler Cartridge is with the WRITER/EDITOR. The purpose of the WRITER/EDITOR is to receive and retain what you enter on the Keyboard and to follow your instructions to modify what you previously wrote.

Poor

1. CALLING THE WRITER/EDITOR AND OTHER CONTROL PROGRAMS

In order to use the WRITER/EDITOR(or any other cartridge program) it must be called, or "invoked". That is, the program that will control the communication between you, at the keyboard, and the TV screen, must be the WRITER/EDITOR and not some other program.

Only one of five programs (four if you do not have a diskette drive) can be in control if you are using the Assembler Cartridge. They are the WRITER/EDITOR, DEBUGGER, ASSEMBLER, DOS and MEMO-PAD. ~~The MINI-ASSEMBLER, accessible only from the DEBUGGER, could be considered a sixth program.~~ The keyboard entries that will transfer control from one of these to another are shown in Figure 3.

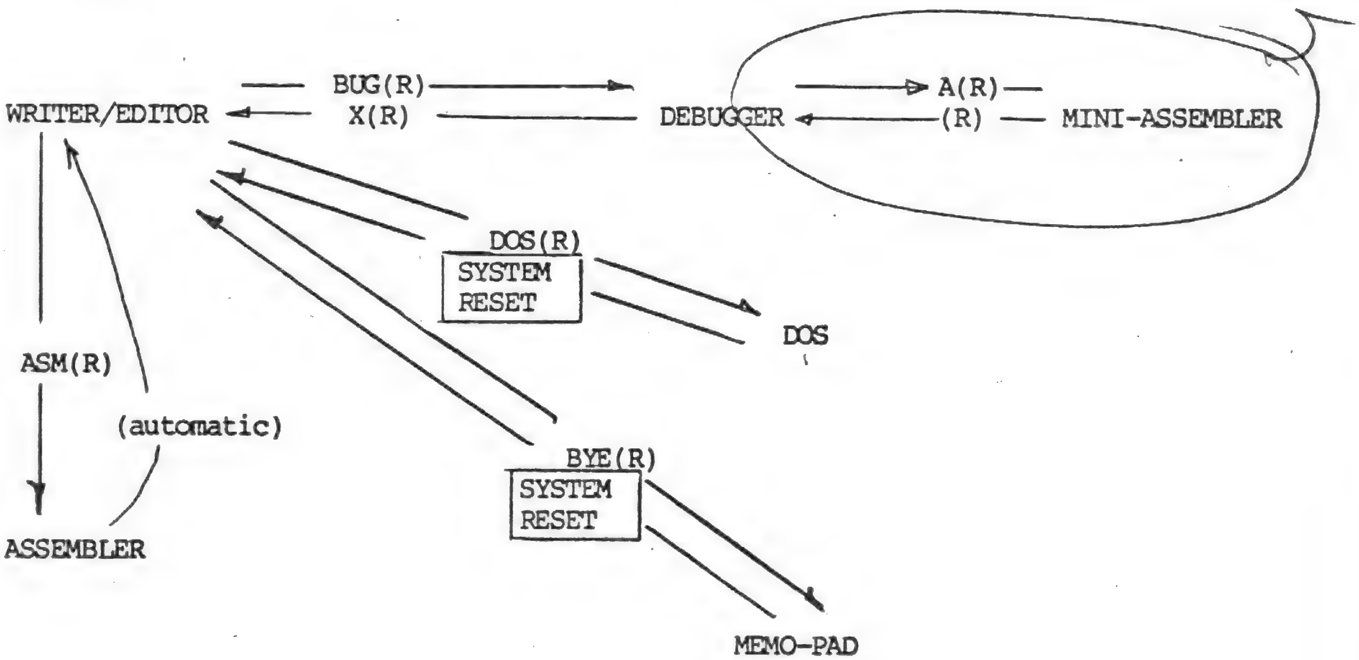


Figure 3. The control programs possible with the Assembler Cartridge, and how to transfer from one to another. "(R)" signifies RETURN.

MEMO PAD

MEMO-PAD is the name given to the control mode when no cartridge is installed. Therefore, when you see "MEMO-PAD" on the screen and you cannot get out of it by depressing the system reset key, you have either failed to insert the Assembler Cartridge or you have not properly seated the cartridge in its socket.

If the cartridge is properly inserted, you can transfer into MEMO PAD by typing BYE (RETURN). You can transfer from MEMO-PAD back to WRITER/EDITOR by depressing the SYSTEM RESET key.

All the screen-editing functions operate in MEMO-PAD, so you can compose the screen. The text on the screen is stored in a temporary buffer and cannot be transferred to a permanent storage medium. Since the screen in MEMO PAD is kept independent of any program you are writing in the Write/Edit mode, MEMO PAD is a convenient mode to use when you must interrupt a programming task. You can leave a message on the screen as a reminder to yourself or as information for others. For example,

```
ATARI COMPUTER - MEMO PAD

GONE TO LUNCH. BACK AT 1.15

PLEASE DO NOT ERASE[]
```

1.B.6

only

DOS

DOS is the Disk Operating System used with an Atari computer having at least 16k of memory. It is supplied with the Atari Diskette Drive. The DOS is written on a system diskette. Its function is to manage data transfers to and from a diskette on which you wish to save data. When the diskette drive is powered up and the system diskette is loaded, powering the computer will cause the DOS to be loaded into RAM. Some of the DOS facilities can be used by the WRITER/EDITOR to permit access (read/write) to diskette. When you transfer to DOS, by entering DOS (RETURN), you can read/write on diskette independently of the Assembler Cartridge. For a detailed understanding of DOS, refer to the DOS manual supplied with the diskette drive.

DEBUGGER and MINI-ASSEMBLER

The DEBUGGER, entered by typing BUG (RETURN), has many commands that enable you to examine details of your assembled program. From the DEBUGGER you can enter the MINI-ASSEMBLER, which allows you to write instructions in Assembly Language and have each instruction assembled immediately.

WRITER/EDITOR and ASM

It is clear from Figure 3 that when using the Assembler Cartridge you are normally in the Write/Edit mode or the Debug mode. When the Atari 800 is powered up with the Assembler Cartridge installed, control is automatically transferred to the Write/Edit mode. The screen will show this by writing EDIT on the top line and the cursor, a solid square filling one character space, on the next line.

You transfer into the Assemble mode only from the Write/Edit mode (type the command ASM) and the return from the Assemble mode is automatic when assembly of the program is complete.

The Assemble command (ASM) has three parameters not shown in Figure 3. If you do not specify the parameters, the Assembler assumes standard values. The ASM command is more fully described in the section on Saving and Retrieving Commands.

1.b.6

1.a.6 only

Having powered up your 400 or 800 with the Assembler Cartridge installed, you should see on your screen the message

EDIT

■

With your diskette drive connected you will have to wait a few seconds while DOS is loaded from diskette into RAM. If you have the diskette drive connected and powered, but you have no diskette running, the system will be unsuccessful in attempting to boot the diskette drive. Consequently, you will rec eive the normal Boot Error message on the screen. You can avoid this by pressing the SYSTEM RESET key, and the EDIT message will appear immediately. You can also prevent the system from attempting to boot the diskette drive by either disconnecting the drive from the consol or by powering down the drive.

Once the EDIT message appears, any entries on the keyboard will appear in successive locations across the screen, starting at the cursor ([]) position. The screen is 36 locations wide, and there is an automatic "wraparound" when you come to the end of a line, so that your entries automatically continue on the next line. upto N chars 3 lines

The EDIT message tells you that you are in the Write/Edit mode. If you do not see the EDIT message, some error or malfunction has occurred. You should check the Operators Manual to make sure that you have followed procedures correctly.

1.b.17

2. GETTING STARTED

In this section we explain three "getting started" commands, namely SIZE, LOMEM and *=.

Certain decisions have to be made before a single Assembly Language instruction is assembled. These decisions have to do with where various aspects of the information to be generated will be stored. The decisions are already constrained by the demands of the rest of the system. We have to explain how the system handles your input from the keyboard.

Your input from the keyboard goes to two locations:

1. CURRENT LINE BUFFER. This is a short buffer of 108 locations in which the characters of the current line, up to a Return, are stored in ATASCII form. The line can be 106 characters long, including the Line Number at the beginning of the line. The buffer is overwritten by each successive line. It is located in a scratch-pad area of 484 locations (180 hex) reserved for the Assembler Cartridge.

2. EDIT TEXT BUFFER. This buffer starts above the Assembler scratch-pad area. It stores, in ATASCII form, all the input from the keyboard. The information displayed on the screen when you are writing a program comes from the Edit Text Buffer. When you alter a line on the screen, by using the keyboard editing functions and RETURN, you also alter what is stored in the Edit Text Buffer.

As you enter more and more on the keyboard, the Edit Text Buffer fills up. There is no upper bound specified. However, it is possible to attempt to make the Edit Text Buffer extend into non-existent memory or into memory reserved for the OS.

The remaining locations between the top of the Current Line Buffer and the bottom of the Edit Text Buffer are scratch-pad RAM used by the Assembler Cartridge.

SIZE Command

To find the location of the Current Line Buffer and the Edit Text Buffer, type the Command SIZE. Three hex numbers will be displayed, as follows:

```
EDIT
SIZE (RETURN)
0700                     0880                     5C1F

EDIT
[]
```

The first number (0700 in this example) is the start of the Current Line Buffer. The middle number (0880) is the start of the Edit Text Buffer. These numbers are larger when DOS is loaded from a system diskette on power-up.

If SIZE is the first command (and you have not entered anything else on the keyboard), then the second number will be 180 (hex) larger than the first. The second number gets larger, however, as you enter text on the keyboard. The second number is actually the address where the next character you type will be stored.

You can find out at any time how long your program is with the SIZE command. Subtract the first number and 180 from the second number. The result is approximately the number of characters in your program.

The third number (5C1F) is the highest available memory location. It depends on the type and quantity of memory boards in your computer.

Thus, all three numbers differ according to the configuration of your system. However, they are always the same (at the start of a program) until you change them, as described next.

LOMEM Command

You can change the location of the buffer area by the LOMEM Command. The locations of the Assembler scratch pad, including the Current Line Buffer, and the Edit Text Buffer, are moved by a single command. The command has the form LOMEMxxxx. For example, the command

```
LOMEM1234   (RETURN)
```

changes the Current Line Buffer (the start of the scratch-pad area) to 1234 hex and the start of the Edit Text Buffer to 13B4 hex (1234 + 180). You can verify this by entering a SIZE Command after the LOMEM Command.

You must not choose a LOMEM address lower than the lowest address shown by the SIZE Command. If you did choose a lower address, your buffers would write over some of the memory reserved for the Operating System. The lowest address that the Command LOMEM will show is 0700 (hex).

One example of the use of the LOMEM Command is to reserve memory in the area of the normal (default) location of the Assembler scratch pad. Consider the following commands:

```
SIZE   (RETURN)
0700            0880            5C1F
EDIT
▯
        (POWER OFF, POWER ON)
EDIT
LOMEM 800
EDIT
▮
```

The effect of the LOMEM Command in this example is to permit you to use in your assembled program the 256 locations from 0700 to 0800.

You cannot change the location of the buffers after you have started writing a program. If you use LOMEM, it must be the first command after you power up. That is why power was switched off and on after the SIZE Command in the example, above.

*= Command

By using the SIZE command you know where your program, as entered on the keyboard, will be stored, but you must also specify where the program, as translated into machine language, will be located. The Assembler must know where the machine language equivalent of the very first instruction must be located. Therefore, a starting location must be specified before the first Assembly Language instruction. This is done with the *= Command. The command corresponds to ORG in some other assemblers. The *= Command must have a Line Number followed by a least 2 spaces to be out of the label field. It must be written as shown, without a space between * and =.

Example:

```
EDIT
10  *=$5000
20  TAX
30  ...
```

The effect of line 10 is exerted when the program is assembled. The effect is to place the number AA (the code for TAX) in location $5000, and subsequent instructions in successive locations.

The *=Command should appear before the first Assembly Language instruction, but it can also be used anywhere in your program to change the Program Counter. For example, to reserve four locations after line 120, write:

```
121  *=*+4  (RETURN)
```

The initial *=Command may actually be the one that is written last. After you have finished writing your program, you may determine the top of your Edit Text Buffer with a SIZE Command. For example,

```
SIZE    (RETURN)
2800              346F        9C1F

EDIT
█
```

That tells you that it is safe to assemble your program between 3470 and 9C1F, so that you might then add to your program line 0.

```
0  *=$3500     (RETURN)
```

In this fashion you add to the beginning of your program a statement that guarantees that the ASSEMBLER does not overwrite your source program. The difference between $3470 and $3500 gives you a margin for any minor debugging changes that increase the length of your source program.

CAUTION If you neglect to include a *= Command, the ASSEMBLER will attempt to assemble your program starting at address 0000. Your program will start to be assembled in Operating System RAM, changing data that the Operating System needs to function properly.

Sample Program

We are now ready to give a small sample program to show the LOMEM, SIZE and *= Commands, the recommended spacing when you enter your program on the keyboard, and the spacing of the assembled program on the screen.

Figure 4 represents the appearance of the screen as you enter your program and after you give the command to assemble (ASM). Only 24 lines are shown at a time on the screen, so everything shown in Figure 4 will not be on the screen at the same time.

```
EDIT
LOMEM5000

EDIT
SIZE
5000            5180            7C1F

EDIT
10  *=$3000
20 REP LDY ABSY
30  BEQ YEQ SAME PAGE
40  INY
50  JMP REP
60 ABSY=$3744
70 YEQ=*+$60
90  .END
ASM


               10              *=   $3000
3000 AE4437    20 REP          LDY  ABSY
3003 F064      30              BEQ  YEQ
.. SAME PAGE
3005 C8        40              INY
3006 4C0030    50              JMP  REP
3744           60 ABSY         =    $3744
3069           70 YEQ          =    *+$60
               90              .END

EDIT
█
```

Figure 4. Appearance of the screen as your program is entered (above) and assembled (below).

Referring to Figure 4, note the following points:

1. The Commands LOMEM, SIZE and ASM do not have Statement Numbers.

2. All instructions and Directives have Statement Numbers.

3. A Label starts after one blank space following the Statement Number (e.g., Statement 20).

4. An instruction or Directive starts after two blank spaces following the Statement Number (e.g., Statements 10, 30, 40, 50, 80)

5. Single spaces separate Label, Op Code Mnemonic, Operand and Comment (e.g., lines 20 and 30).

6. Using Statements 20 and 30 as examples, the format of the assembled program is shown below. Note, however, that some of the spacing can be change by the TAB Directive (q.v.).

```
3000 AE4437 20 REP     LDX   ABSX
3003 F064      30      BEQ   XEQ
      SAME PAGE
```

Operand

Op Code Mnemonic

Label

Statement Number

Instruction

Comment from previous line starts here ~~on next line~~

Address

3. COMMANDS TO EDIT A PROGRAM

Now that we have explained how to get started writing a program, it is up to you actually to write the program. This manual contains very little information on Assembly Language programming techniques. We assume that you are skilled in writing Assembly Language. The remainder of this section describes Editing Commands, not Assembly Language instructions.

Commands and instructions are distinguished. An instruction has a Line Number; a Command has no Line Number and is executed immediately. We did refer to "*=" as a Command, even though it is not executed immediately and it must have a Line Number. It has the character of an obligatory Directive.

NEW Command

This Command clears the Edit Text buffer. After this Command you cannot restore any program that you had in the buffer.

LIST Command

This Command displays on the screen the program in the Edit Text buffer.

LIST (RETURN) displays the whole program.

LISTxx (RETURN) displays only line xx.

LISTxx,yy (RETURN) displays line xx to yy.

Using our sample program as an example:

```
EDIT
LIST (RETURN)

10  *=$3000
20 REP LDX ABSX
30  BEQ XEQ SAME PAGE
40  INX
50  JMP REP
60 ABSX=$3744
70 XEQ=*+$60
80  .END

EDIT
LIST20  (RETURN)

20 REP LDX ABSX

EDIT
LIST 50,70 (RETURN)

50  JMP REP
60 ABSX=$3744
70 XEQ=*+$60

EDIT
```

PRINT Command

This command is the same as LIST, except that it prints Statements without Statement Numbers.

Example:

```
EDIT
PRINT (RETURN)
  *=$3000
 REP LDX ABSX
  BEQ YEQ SAME PAGE
  INX
  JMP REP
 ABSX=$3744
 YEQ=*+$60
  .END

■ (RETURN)
```

After using a PRINT Command, no further command can be entered until you press RETURN, which causes the EDIT message and cursor to be displayed.

```
EDIT
PRINT 20 (RETURN)

 REP LDX ABSX
■ (RETURN)
```

After using a PRINT Command, no further command can be entered until you press RETURN, which causes the EDIT message and cursor to be displayed.

```
EDIT
PRINT 40,60
  INX
  JMP REP
 ABSX=$3744
■ (RETURN)

EDIT
■
```

After using a PRINT Command, no further command can be entered until you press RETURN, which causes the EDIT message and cursor to be displayed.

FIND Command

This Command finds a specified string. The ways to write the Command are shown below.

FIND/SOUGHT/ (RETURN)	finds the first occurrence of the string "SOUGHT". The Statement that contains the string is displayed.
FIND/SOUGHT/xx (RETURN)	finds the string "SOUGHT" if it occurs in Statement Number xx. Statement xx is displayed if it contains the string.
FIND/SOUGHT/xx,yy,A	Finds all occurrences of the string "SOUGHT" between Statement Numbers xx and yy. All the Statements that contain the string are displayed.

DEL Command

This Command deletes Statements from the Edit Text buffer. source pg

DELxx (RETURN)	deletes Statement Number xx.
DELxx,yy (RETURN)	deletes Statement Numbers xx to yy.

I/10/26/79/D.H./26

NUM Command

This Command assigns Statement Numbers automatically.

NUM (RETURN) — increments Statement Number by 10 after each RETURN. The new Statement Number, followed by a space, is automatically displayed.

NUMxx (RETURN) — has the same effect as NUM, but the increment is xx instead of 10.

NUMyy,zz (RETURN) — forces the next Statement Number to be yy and the increment to be zz.

(RETURN) — cancels the NUM Command.

CAUTION You cannot use the special keyboard editing keys to change other Statements while the NUM Command is in effect. You will succeed in changing what appears on the screen, but, in an exception to the generaL rule, the contents of the Edit Text Buffer will not be changed.

RE

REN Command

This Command renumbers Statements in the Edit Text Buffer.

REN (RETURN) renumbers all the Statements in increments of xx, starting with 10.

RENyy,zz (RETURN) renumbers all the Statements in increments of zz, starting with yy.

why zz instead of xx

REP Command

This Command replaces a specified string in the Edit Text Buffer with a different specified string.

REP/OLD/NEW/ (RETURN)	replaces the first occurrence of the string "OLD" with the string "NEW".
REP/OLD/NEW/xx,yy (RETURN)	replaces the first occurrence of the string "OLD" between Statements Number xx to yy with the string "NEW".
REP/OLD/NEW/,A (RETURN)	replaces all the occurrences of the string "OLD" with the string "NEW".
REP/OLD/NEW/xx,yy,A (RETURN)	replaces all the occurrences of the string "OLD" between Statements xx to yy with the string "NEW".
REP/OLD/NEW/xx,yy,Q (RETURN)	displays, in turn, each occurrence of the string "OLD" between Statements Number xx and yy. To replace the displayed "OLD" with "NEW", type Y. To retain the displayed "OLD", press RETURN.

(does Y require return)

4 COMMANDS TO SAVE AND RETRIEVE

Commands associated with saving and retrieving a program or a block of memory are:

$$\left.\begin{array}{l}\text{LIST}\\ \text{ENTER}\\ \text{ASM}\end{array}\right\}\text{ a program}\qquad \left.\begin{array}{l}\text{SAVE}\\ \text{LOAD}\end{array}\right\}\text{ a specific block of memory}$$

These commands are similar in the way that their parameters are specified. They are also similar in that they cause data transfers between memory and long-term store. We distinguish, however, between commands that manipulate complete programs (LIST, ENTER, ASM) and commands that address specified memory locations. (SAVE, LOAD).

LIST has already been described in one use, as an editing command, which is a special case of a more general concept.

We have also mentioned ASM before, in Figure 3 and the section on WRITER/EDITOR and ASM. Its use is more versatile than simply calling the Assembler, as will be shown in this section.

The commands ENTER, LOAD and SAVE have not been mentioned before. The commands LIST and ENTER are respectively for saving and retrieing programs, and they correspond to the commands to SAVE and LOAD a specific block of memory.

While you are still developing a program, over several distinct sessions, you will have more occasion to use the LIST, ENTER and ASM commands. When your source program is satisfactory you will more often be working with the object program, using the SAVE and LOAD commands.

wording

LIST Command

The command LIST is used to save programs. If no device is specified in the command, the device "screen" is assumed. Possible devices are the TV screen (#S:), the Printer (#P:) the cassette (#C:) and the diskette (#D:). The commands to transfer a program to these various devices are:

LIST#S:	(LIST#S: is the same as LIST)
LIST#P:	
LIST#C:	(CAUTION: Use cassette-handling procedures described in Users Manual)
LIST#D:NAME	where "NAME" is an arbitrary name you give to the program. NAME must start with a letter and have no more than 8 characters, consisting of letters and numbers only. It may also have an extension of up to 3 characters. For example, NAME3, ST5, JOHN. 23, PROP.13 are all legal names.

The forms of the commands to transfer only lines xx to yy to a device are:

LIST~~$~~#S:,xx,yy	(LIST#S:,xx,yy is the same as LIST,xx,yy)
LIST#P:,xx,yy	
LIST#C:,xx,yy	(CAUTION: Use cassette-handling procedures described in Users Manual)
LIST#D:NAME,xx,yy	where "NAME" is an arbitrary name you give to the program. See the description above.

CAUTION: The DOS makes sure that every file has a unique name by deleting old files if necessary. Therefore, do not name a file you are listing to diskette with the name of a file that is already stored on the diskette, unless you wish to replace the existing file with the one you are listing.

I/10/26/79/D.H./30

ENTER Command

The Command ENTER is used to retrieve programs. As with the Command LIST, a device has to be specified, in this case the device where the program is stored. There is only one device, the diskette, on which a named program is stored in a retrievable form. To retrieve a source program from diskette, the command is:

ENTER#D:NAME where "NAME" is the arbitrary name you gave to the program when you listed it on the diskette.

To retrieve a source program from cassette, the command is:

ENTER#C: (Follow the cassette-handling procedures give in your Users Manual.)

Commands associated with saving and retrieving the contents of specific memory locations, rather than programs in the Edit Text Buffer, are SAVE and LOAD. They correspond to LIST and ENTER, respectively.

Good for saving object?
how do you make it bootable?

SAVE Command

The contents of a block of memory, locations xxxx to yyyy, on cassette or diskette, the commands are:

SAVE#C: xxxx,yyyy

SAVE#D:NAME xxxx,yyyy where NAME is an arbitrary name that you give to the block that you are saving.

LOAD Command

To retrieve the contents of the block of memory that you SAVED and which you named NAME the command is:

LOAD#C: (CAUTION: Use the cassette-handling procedures described in your Users Manual.)

LOAD#D:NAME where "NAME" is the arbitrary name that you gave to the block when you saved it on diskette.

These commands will reload the memory locations xxxx to yyyy with the contents that were previously SAVEd. The numbers xxxx and yyyy are those that were given in the original SAVE Command.

ASM Command

ASSEMBLE COMMAND

We now deal with the Assemble command, ASM. To assemble a program type

ASM (RETURN)

The Assembler then translates the source program in the Edit Text Buffer, line by line in no more than two passes, into machine code, writes the machine code into the memory locations that are specified in the source program and displays the assembled program on the TV screen. In transferring data between parts of the system, the Assembler uses Operating System facilities.

In this example we specified no particular device where the source program would be found and no particular device where the program was to be listed or assembled. These are variables of the ASM command. When we do not specify the value of these variables, the Assembler proceeds with the standard or "default" values. That is, it finds the source program in the Edit Text Buffer, lists the assembled program on the TV screen and assembles the program into memory. You can override these standard decisions in the ASM command.

The form of the command is, in general:

ASSEMBLE (Device where Source program located), (Device where assembled program will be listed), (Device where Object Code will be stored)

The form of the command, as written, is:

$$\text{ASM}\ \left\{\text{\#D:NAME1}\right\},\ \left\{\begin{matrix}\text{\#D:} \\ \text{\#P:}\end{matrix}\right\},\ \left\{\begin{matrix}\text{\#D:NAME2} \\ \text{\#P:}\end{matrix}\right\}$$

D Diskette
P Printer

The form in which the command is written above is intended to convey that you will type one alternative from each set of braces. For example:

ASM#D:LISTED,#P:,#D:OBJ

That command takes the source program that you had previously listed on diskette and called "LISTED", assembles it and lists the assembled form on the Printer, and records on the diskette, with the new name "OBJ", the machine code translation of the source "LISTED". Note that commands of this form store the machine code on the diskette and not in computer memory.

Some other combinations are possible, but not useful. Some of the combinations shown are not useful either. If you attempt an illegal combination, you will be given an error message.

II/10/26/79/dh/3.part 1

To make a standard selection, enter a comma, as in the following useful command

ASM,#P:

That command takes the source program from the default Edit Text Buffer, assembles and lists it on the Printer as before, and stores the machine code directly into the computer memory.

CAUTION: You must be careful to specify a "safe" starting address at which to load the machine code into memory before executing the above command. (See *= Command). If you try to assemble the program into Operating System memory, you will probably cause an Operating System failure. If that happens and you have to switch off your computer, you will, of course, lose the program in the Edit Text Buffer, and you will have to enter it again.

In the next example only the final parameter is specified, and the first two have default values:

ASM,,#D:OBJ2

That command takes the source program from the Edit Text Buffer, lists it on the screen and records on diskette the machine code object program under the name "OBJ2". You can later retrieve OBJ2 from diskette with the following command

LOAD #D:OBJ2

The effect of the command is to load the appropriate memory locations, as specified in the original source program, using the assembled object program on the diskette as the source of the information.

5 DIRECTIVES (PSEUDO OPERATIONS)

Directives are instructions to the Assembler. Directives do not, in general, produce any assembled code, but they affect the way the Assembler assembles other instructions during the assembly process. Directives are also called pseudo operations or pseudo ops.

Directives are identified by the Assembler by the "." at the beginning. The only exception is the = Directive, which starts with a Label or *. You will remember that in the section on getting started we called the command "*=" an obligatory Directive.

A Directive must have Line Numbers, which it follows by at least two spaces.

OPT Directive

This Directive specifies an option. There are four sets of options. These are:

```
.OPT NOLIST
.OPT LIST      (this one is the default condition)

.OPT NOOBJ
.OPT OBJ       (this one is the default condition)

.OPT NOERR
.OPT ERR       (this one is the default condition)

.OPT NOEJECT
.OPT EJECT     (this one is the default condition)
```

1.b.9

The second listed of each pair represents the standard or default condition. Therefore, an option pair, when used, always appears in the program in the order shown, with, for example, .OPT NOOBJ occurring before .OPT OBJ. The first member of each pair might also occur on its own, without the default option being re-imposed before the end of the program.

100 .OPT NOLIST (part of source program) 200 .OPT LIST	The effect of these Directives is to omit from the listed form of the assembled program the lines between lines 100 and 200. (These line numbers are arbitrary.)
100 .OPT NOOBJ (part of source program) 200 .OPT OBJ	The effect of these Directives is to omit from the machine code form of the assembled program the lines between lines 100 and 200.
100 .OPT NOERR (part of source program) 200 .OPT ERR	The effect of these Directives is to omit error messages for the assembled program lines between lines 100 and 200.
100 .OPT NOEJECT (part of source program) 200 .OPT EJECT	The effect of these Directives is to suppress, between lines 100 AND 200, the 4-line page spacing that is normally inserted after every 56 lines of the listed form of the assembled program.

1.b.10

TITLE AND PAGE DIRECTIVES

We explain these Directives together because they are intended to be used together to provide easily read information about the assembled program.

These Directives are most useful when the assembled program is listed on the Printer.

TITLE and PAGE allow you to divide your program listing into segments that ~~are named for your convenience~~ bear messages written for your own convenience, much as you might add short explanatory notes to any complex material.

The PAGE Ditective causes the Printer to put out 4 blank lines, followed by the messages you have given for TITLE and PAGE, ~~followed by another blank line~~. This camses the messages to stand out somewhat from the rest of the assembled program listing. The TITLE message is always the same, and the PAGE message can be different on every occurrence.

You would usually have only one TITLE Directive, giving, say, the program name and date, and different PAGE Directives for giving different page messages. Then, On listing the assembled program the same TITLE message on every page would be followed by the ~~same,~~ a different ~~while the~~ Page message ~~would differ~~.

The blank Printer lines ~~on the Printer~~ that the PAGE Directive produces ~~on the Printer~~ can be used to break up a long printout into segments that can be mounted in a notebook.

To illustrate, let us add to our small program examples of the Directives TITLE and PAGE.

```
10  .TITLE "TEST
           ROUTINE"
20  .PAGE "CHECK FOR ZERO"
30  *=$3000
40 REP LDX ABSY
50  BEQ YES SAME PAGE
60  INX
70  JMP REP
80  .PAGE"ADDRESSES NEEDED BY CHECK"
90 ABSY=$3744
0100 YES=*+$60
0110  .END
```

ASM,#P:

```
             10          .TITLE "TEST
                                ROUTINE"



TEST
ROUTINE
CHECK FOR ZERO

             20          .PAGE "CHECK FOR
 ZERO"
             30          *=   $3000
3000 AE4437 40 REP       LDX  ABSY
3003 F064    50          BEQ  YES        S
AME PAGE
3005 E8      60          INX
3006 4C0030 70           JMP  REP


TEST
ROUTINE
ADDRESSES NEEDED BY CHECK

             80          .PAGE"ADDRESSES
NEEDED BY CHECK"
3744         90 ABSY     =    $3744
3069         0100 YES    =    $+$60
             0110        .END
```

TAB DIRECTIVE

The TAB Directive sets the fields of the Statement as they appear when assembled and listed on the screen or the Printer. Let us use the specific example of Statement 30 of the small sample program we used for illustration before. It was written as follows:

```
20  ...
30  BEQ XEQ SAME PAGE
40  ...
```

Note that one space, rather than a tab, is used between each field. Using spaces rather than tabs lets you write longer programs, since the Edit Text Buffer will not be filled up with the extra spaces that tabs would require.

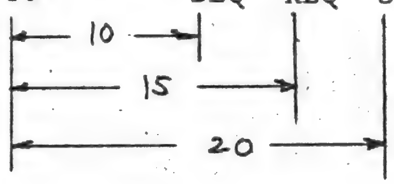
Compressing the program in this way makes it less easily readable than we might wish, but we can use the TAB Directive to give us a more readable assembled version. The form of the Directive is

```
xxx  .TAB 10,15,20
```

The effect of the TAB Directive of line xxx is confined to the appearance of lines following xxx when they are assembled and listed on the Printer or screen.

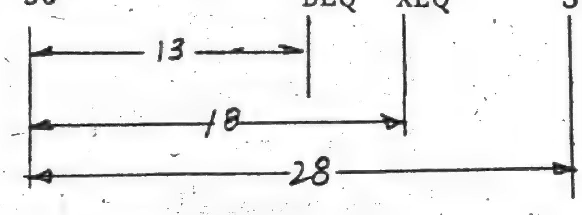
In the case of line 30, the appearance on the Printer would be as shown below:

3003 F064   30   BEQ XEQ SAME PAGE



If the TAB Directive is not used, then the appearance of the assemblec line on the Printer will be as shown below:

3003 F064   30   BEQ XEQ   S
AME PAGE



The appearance of this line on the screen will be different only because the screei has 36 characters positions, while the Printer has 40.

BYTE, DBYTE AND WORD Directives

These Directives are similar in that they are used to insert data rather than instructions into the proper places in the program. Each Directive is slightly different in the manner in which it inserts data.

BYTE Directive

The BYTE Directive reserves a location (at least one) in memory. The Directive essentially increments the Program Counter to leave space in memory to be filled by information required by the program. The Operand supplies the information directly.

Examples:

```
10    .....
20    .BYTE 34
30    .....
```

Here,the Assembler assembles into successive locations the instruction of line 10, then the number 34, then the instruction of line 30.

1.b.20

```
10    .....
20    .BYTE 34,56,78
30    .....
```

Here, the Assembler assembles into successive locations the instruction of line 10, then the numbers 34, 56 and 78, then the instruction of line 30. The Operand may be an expression more complex than the numbers used in the examples. The rules for writing and evaluating an expression are given in Appendix D.

```
10    .....
20    .BYTE "ATARI"
30    .....
```

Here, the Assembler assembles into successive locations the instruction of line 10, then the (ATASCII) hex numbers 41, 54, 41, 52 and 49, then the instruction of line 30.

DBYTE DIRECTIVE

The DBYTE Directive reserves two locations for each expression in the operand. The value of the expression is assembled with the high-order byte first (in the lower numbered location). For example:

```
10   *=$4000
20   .DBYTE   ABS+$3000
```

When line 20 is assembled and the value of ABS + $3000 is determined to be (say) 5123 ,51 is put in location 4000 and 23 is placed put in location 4001.

WORD DIRECTIVE

The Word Directive is the same as the DBYTE Directive except that the value of the expression is stored with the low-order byte first. For example:

```
10   *=$4000
20   .WORD  ABS*$3000
```

When line 20 is assembled and the value of ABS + $3000 is determined, as before, to be 5123, 23 is put in location 4000 and 51 is put in location 4001.

The WORD Directive simplifies some programming since addresses in machine code are always given in the order low-byte followed by high byte. Therefore, the WORD Directive is useful, for example, constructing a table of addresses.

= Directive

The = Directive is used to give a value to a label. Two examples appear in the sample program we used before. Statements 60 and 70 give values to ABSX and XEQ, as follows:

```
60 ABSX=$3744
70 XEQ=*+$60
```

Since the symbol that is given a value is a Label, there must be only one space after the Statement Number. The expression on the right cannot have a value greater than FFFF (hex). The rules for writing and evaluating an expression are given in Appendix D.

When the = Directive is used to give a value to a Label, the Label can be used in an Operand, by itself, as in statements 20 and 30 in the sample program.

A defined Label may also appear as part of an expression. Our sample program does not contain an example, so we give one below, in line 240

```
100 TAB1=$3000
..............
..............
..............

240 TAB2=TAB1+$20
```

When the program is assembled, TAB2 will be given the value $3020.

You should note that defining a label in this way gives the label a specific address; it does not define the contents of the address. In the example, above, TAB1 and TAB2 might be the location of two tables that contained the values of variables that your program required.

We also encountered the = Directive in the Getting Started Commands, where it is used to set the starting location of the assembled program. In that case we defined the value of the Current Location Counter (*), rather than a Label. Minimal statements of the two uses of the = Directive are given below. Note that there must be exactly one space before AB and at least two spaces before *.

```
10 AB=Expression
10  *=Expression
```

END DIRECTIVE

Every program should have one and only one END Directive. It tells the Assembler to stop assembling. It is not always essential, but it is good practice to use this Directive.

DEBUGGING.

1. PURPOSE OF DEBUGGER

The Debugger allows you to follow the operation of a program in detail and to determine the effects of minor changes in it.

The operation of the program is shown in both machine language and Assembly Language. A knowledge of machine langage is helpful when you use the Debugger, but it is not essential. The Debugger is able to convert machine code into Assembly Language (disassemble), so you can make code alterations at the level of particular memory locations and be informed of the effects by the Debugger in Assembly Language.

2. CALLING THE DEBUGGER

The Debugger is called from the WRITER/EDITOR by typing

BUG (RETURN)

This produces on the screen

DEBUG

█

The command to return to the WRITER/EDITOR is

X (RETURN)

3. DEBUG COMMANDS

The Debug commands are listed below. In this list "xxxx" indicates that the form of the command must include memory adress(es). The actual method of specifying the memory adress(es) is shown in the examples given later in this section.

DR	Display Registers
CR	Change Registers
Dxxxx	Display Memory
Cxxxx	Change Memory
Mxxxx	Move Memory
Vxxxx	Verify Memory
Lxxxx	List Memory
Axxxx	Assemble Memory
Txxxx	Trace Operation
Sxxxx	Single-step Operation
Gxxxx	Go (Execute Program)
X	Return to WRITER/EDITOR
(BREAK)	Pressing the BREAK key halts certain operations.

We now give several examples showing how to use the commands. In the examples, the lines ending with "(RETURN)" are entered on the keyboard. The other lines show the response of the system, as displayed on the screen.

DR Display Registers

Example:

```
EDIT
BUG (RETURN)

DEBUG
DR (RETURN)
    A=BA   X=12   Y=34   P=B0   S=DF
DEBUG
■
```

The registers and contents are displayed as shown. A is the Accumulator, X and Y are the Index Registers, P is the Processor Status Register and S is the Stack Pointer.

CR Change Registers

Example:

```
EDIT
BUG (RETURN)

DEBUG
CR<1,2,3,4,5 (RETURN)

DEBUG
■
```

The effect of the command above is to set the contents of the Registers A, X, Y, P and S to 1, 2, 3, 4 and 5.

You can skip Registers by using commas after the '<'. For example,

```
CR<,,,,E2 (RETURN)
```

sets the Stack Pointer to E2 and leaves the other Registers unchanged. Registers are changed in order up to RETURN. For example,

```
CR<,34 (RETURN)
```

sets the X Register to 34 and leaves the other Registers unchanged.

Dxxxx Display Memory

Dxxxx,yyyy where yyyy where yyyy< xxxx shows the contents of address xxxx.

Example:

```
DEBUG
D5000,0  (RETURN)

5000  A9
DEBUG
■
```

This shows that address 5000 contains the number A9.

If the second address (yyyy) is omitted, the contents of 8 successive locations are shown. The process can be continued by typing D (RETURN).

Example:

```
DEBUG
D5000  (RETURN>

5000  A9  03  18  E5  F0  4C  23  91
DEBUG
D  (RETURN)

5008  18  41  54  41  52  49  20  20
DEBUG
■
```

Dxxxx,yyyy where yyyy>xxxx, shows the contents of addresses xxxx to yyyy.

Example:

```
DEBUG
D5000,500F  (RETURN)

5000 A9  03  18  E5  F0  4C  23  91
5008 18  41  54  41  52  49  20  20
DEBUG
■
```

The display can be stopped by pressing the BREAK key.

Cxxxx Change Memory

Cxxxx<yy changes the contents of address xxxx to yy

Example:

```
DEBUG
C5001<23  (RETURN)

DEBUG
█
```

The effect of the command is to put the number 23 in location 5001,

A comma increments the location to be changed.

Example:

```
DEBUG
C500B<21,EF  (RETURN)

DEBUG
C700B<31,,,87  (RETURN)

DEBUG
█
```

The first command puts 21 and EF in locations 500B and 500C respectively.

The second command puts 34 and 87 in locations 700B and 700E respectively.

Mxxxx Move Memory

Mxxxx<yyyy,zzzz copies memory for yyyy-zzzz to memory starting at xxxx. Address must be less than yyyy or greater then 3333.

Example:

```
DEBUG
M1230<5000,500F (RETURN)

DEBUG
█
```

The command copies the data in location 5000-500F to locations 1230-123F.

Vxxxx Verify Memory

Vxxxx<yyyy,zzzz compares memory yyyy-zzzz with memory starting at xxxx and shows mismatches.

```
Example:
DEBUG
V7000<7100,7123  (RETURN)

DEBUG
█
```

The command compares the contents of 7100-7123 with the contents of 7000-7023. There were no mismatches.

Mismatches are shown as follows:

```
7101  00     7001  22
7105  18     7005  10
```

Lxxxx List Memory with Disassembly

This command allows you to look at any block of memory in disassembled form.

Examples:

7000 (RETURN)	List a screen page (20 line of code) starting at memory location 7000. Pressing the BREAK key during listing halts the listing.
L (RETURN)	This form of the command lists a screen page starting at the instruction last shown, plus ~~one~~ 1.
L7000,0 (RETURN) L7000,7000 (RETURN) L7000,6000 (RETURN)	These forms list the instructions at address 7000 only.
L345,567 (RETURN)	This form lists address 345 through 567 Only the last 20 instructions will actually be visible at the completion of the response of the system.

The command Lxxxx differs from Dxxxx in that Lxxxx disassembles the contents of memory.

Example:

```
EDIT
BUG  (RETURN)

DEBUG
L5000,0  (RETURN)
5000          A9 03          LDA #$03

DEBUG
█
```

This example shows that the Debugger examined the contents of memory address 5000 and disassembled A9 to LDA. Since A9 must have a one-byte Operand, the Debugger made the next byte (the contents of address 5001) the operand. Therefore, although the Debugger was only "asked" for the content of location 5000, it showed a certain amount of intelligence and replied by showing the instruction that started at address 5000.

To illustrate this further, the number 03 corresponds to no machine code instruction, so the Debugger would interpret 03 as an illegal instruction, and alert you to a possible error, as shown below.

```
DEBUG
L5001,0  (RETURN)
5001,03          ???

DEBUG
```

However, if the first instruction you wrote were LDA $8A, then you would have obtained the following, apparently inconsistent, results while debugging:

```
DEBUG
L5001,0            A9 8A          LDA #$8A

DEBUG
L5001,0             8A            TXA
```

Because the disassembler starts disassembling from the first address you specify, you have to take care that the first address corresponds to the contains the first byte of a "real" instruction.

A Assemble

The DEBUGGER in has a "miniassembler", that can assemble one instruction at a time. To enter the Assemble mode, type

A (RETURN)

Once in the Assemble mode, you stay there until you wish to return to DEBUGGER, which you may do by pressing RETURN.

To assemble an instruction, first enter the address at which you wish to have the machine code inserted. The number that you enter will be interpreted as a hex address. Now type "<" followed by at least one space, then the instruction.

Example:

```
EDIT
BUG  (RETURN)

DEBUG
A  (RETURN)
5001<   LDY  $1234  (RETURN)
5001 AC3412
5004< INY  (RETURN)
5004 C8
█  (RETURN)

DEBUG
█
```

Since the mini-assembler assembles only one instruction at a time, it cannot refer to another instruction. Therefore, it cannot interpret a Label. Consequently, Labels are not legal in the mini-assembler.

You can use the Directives BYTE, DBYTE AND WORD.

Gxxxx Go (Execute Program)

This command executes instructions starting at xxxx.
For example:

G7B00 (RETURN)	Executes instructions starting at location 7B00. Execution continues indefinitely. Execution is stopped by pressing the BREAK key.

Txxxx Trace Operation

This command has the same effect as Gxxxx, except that after execution of each instruction the screen shows the instruction address, the instruction in machine code, the instruction in Assembly Language (disassembled by the Debugger--not necessarily the same as you wrote it in Assembly Langauge) and the values of Registers A, X, Y , P and S.

The execution stops at a BRK instruction (machine code 00) and when you press the BREAK key on the keyboard.

Example:

```
DEBUG
T5000   (RETURN)
5000         A9    03          LDA      #$03
   A=03      X=02      Y=03      P=34     s=05
5002         18                CLC
   A=03      X=02      Y=03      P=34     S=05
5003         E5    FO          SBC      $FO
   A=03      X=02      Y=03      P=34     S=05
5005         4C    23    71    JMP      $7123
   A=03      X=02      Y=03      P=34     S=05
7123         00                BRK
   A=03      X=02      Y=03      P=B4     S=05
DEBUG
■
```

Sxxxx Step Operation

This command has the same effect as Txxxx, except that only one instruction is executed. To step through a program type S (RETURN) repeatedly after the first command of Sxxxx (RETURN).

X Exit

To return to the WRITER/EDITOR enter
X (RETURN)

Type of Instruction	Mnemonic	Operation	Notes	Immed. :$hh #BY	Abs. $hhhh ABS	Page Ø $hh BY	Abs,X $hhhh,X ABS,X	Abs,Y $hhhh,Y ABS,Y	Ind,X ($hh,X) (BY,X)	Ind,Y ($hh),Y (BY),Y	Ind,X, Page Ø $hh,X BY,X	Ind,Y, Page Ø $hh,Y BY,Y	Register A or X or Y	OP CODE (ADDR. TYPE) Implied (IM), Relative (R), Indirect (IN), Absolute (A), None (Ø)	N	Z	C	I	D	V
	LDA	M → A	(3)(1)	A9	AD	A5	BD	B9	A1	B1	B5				*	*				
	LDX	M → X	(1)	A2	AE	A6		BE				B6			*	*				
	LDY	M → Y	(1)	AØ	AC	A4	BC				B4				*	*				
	STA	A → M			8D	85	9D	99⑤	81	91⑥	95									
	STX	X → M			8E	86						96								
	STY	Y → M			8C	84					94									
				②	④	③	④	④	⑥	⑤	④	④								
Arithmetic & Logical	ADC	A+M+C → A	(1)	69	6D	65	7D	79	61	71	75				*	*	*			*
	AND	A∧M → A	(1)	29	2D	25	3D	39	21	31	35				*	*				
	INX	X+1 → X											E8		*	*				
	INY	Y+1 → Y											C8		*	*				
	INC	M+1 → M			EE⑥	E6⑤	FE⑦				F6⑥				*	*				
	DEX	X−1 → X											CA		*	*				
	DEY	Y−1 → Y											88		*	*				
	DEC	M−1 → M			CE⑥	C6⑤	DE⑦				D6⑥				*	*				
	ORA	A∨M → A		Ø9	ØD	Ø5	1D	19	Ø1	11	15				*	*				
	EOR	A⊻M → A		49	4D	45	5D	59	41	51	55				*	*				
	SBC	A−M−C̄ → A	(1)	E9	ED	E5	FD	F9	E1	F1	F5				*	*	*			*
	CMP	A − M	(1)	C9	CD	C5	DD	D9	C1	D1	D5				*	*	*			
	CPX	X − M		EØ	EC	E4									*	*	*			
	CPY	Y − M		CØ	CC	C4									*	*	*			
				②	④	③	④	④	⑥	⑤	④	④	②							
Shift & Rotate	ASL				ØE	Ø6	1E				16		ØA		*	*	*			
	ROL				2E	26	3E				36		2A		*	*	*			
	LSR				4E	46	5E				56		4A		Ø	*	*			
	ROR				6E	66	7E				76		6A		*	*	*			
					⑥	⑤	⑦				⑥		②							
Compare Bits	BIT	A∧M			2C ④	24 ③									M7	*				M6
Register Transfer	TAX	A → X												AA (IM)	*	*				
	TXA	X → A												8A (IM)	*	*				
	TAY	A → Y												A8 (IM)	*	*				
	TYA	Y → A												98 (IM)	*	*				
	TSX	S → X												BA (IM)	*	*				
	TXS	X → S												9A (IM) ②						
Clear Flags / Set	CLC	Ø → C												18 (IM)			Ø			
	CLI	Ø → I												D8 (IM)				Ø		
	CLD	Ø → D												58 (IM)					Ø	
	CLV	Ø → V												B8 (IM)						Ø
	SEC	1 → C												38 (IM)			1			
	SEI	1 → I												F8 (IM)				1		
	SED	1 → D												78 (IM) ②					1	
Branch	BMI	Branch if N=1												3Ø (R)						
	BPL	" " N=Ø												1Ø (R)						
	BEQ	" " Z=1												FØ (R)						
	BNE	" " Z=Ø												DØ (R)						
	BCS	" " C=1												BØ (R)						
	BCC	" " C=Ø												9Ø (R)						
	BVS	" " V=1												7Ø (R)						
	BVC	" " V=Ø												5Ø (R)						
	JMP	Jump												4C (A) ③ 6C (IN) ⑤						
	JSR	Jump to Subr.												2Ø (A) ⑥						
	RTS	Return fr. Subr.												6Ø (IM) ⑥						
	RTI	Return fr. Interrupt												4Ø (IM) ⑥ ② (See Note 2)	From stack					
Stack	PHA	A → M(S)												48 (IM) ③						
	PHP	P → M(S)												Ø8 (IM) ③						
	PLA	S+1 → S; M(S) → A												68 (IM) ④	*	*				
	PLP	S+1 → S; M(S) → P												28 (IM) ④	From stack					
Mach. Control	BRK	Interrupt												ØØ (Ø) ⑦						
	NOP	No operation												EA (Ø) ②						

A	Accumulator, or contents
X	Register X, or contents
Y	Register Y, or contents
M	Memory location, per EA, or contents
M7	Bit 7 of M
M(S)	Memory location, or contents, via stack
∧,∨,⊻	Logical AND, OR, XOR
P→Q	P is copied to Q; P unchanged

At the head of each column, under TYPE OF ADDRESS, the correct way to write an operand is given, in hex, where "h" represents a hex number, and symbolically, where "BY" and "ABS" represent numbers of 1 byte and 2 bytes, respectively.

The circled number at the foot of a column is the number of machine cycles required for the instruction in that block; exceptions are indicated by the circled numbers after the Op Code.

1. If the page boundary is crossed, the number of machine cycles is one more than shown.

2. If the condition is true and the branch is taken, the number of machine cycles is one more than shown when the branch is to the same page and two more than shown when the branch is to a different page.

3. The Z flag is valid when the operand is in hex, not when the operand is in decimal notation.

2   6

w/ it DISISM didn't need this

OP-CODE TABLE

LSD / MSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	LSD / MSD
0	BRK	ORA IND X				ORA Z Page	ASL Z PAGE		PHP	ORA IMM	ASL A			ORA ABS	ASL ABS		0
1	BPL	ORA IND Y				ORA Z Page X	ASL Z Page X		CLC	ORA ABS Y				ORA ABS X	ASL ABS, X		1
2	JSR	AND IND X			BIT Z Page	AND Z Page	ROL Z Page		PLP	AND IMM	ROL A		BIT ABS	AND ABS	ROL ABS		2
3	BMI	AND IND, Y				AND Z Page X	ROL Z Page X		SEC	AND ABS, Y				AND ABS, X	ROL ABS, X		3
4	RTI	EOR IND X				EOR Z Page	LSR Z Page		PHA	EOR IMM	LSR A		JMP ABS	EOR ABS	LSR ABS		4
5	BVC	EOR IND, Y				EOR Z Page X	LSR Z Page X		CLI	EOR ABS, Y				EOR ABS, X	LSR ABS, X		5
6	RTS	ADC IND X				ADC Z Page	ROR Z Page		PLA	ADC IMM	ROR A		JMP IND	ADC ABS	ROR ABS		6
7	BVS	ADC IND, Y				ADC Z Page X			SEI	ADC ABS, Y				ADC ABS, X			7
8		STA IND X			STY Z Page	STA Z Page	STX Z Page		DEY		TXA		STY ABS	STA ABS	STX ABS		8
9	BCC	STA IND, Y			STY Z Page X	STA Z Page X	STX Z Page Y		TYA	STA ABS, Y	TXS			STA ABS, X			9
A	LDY IMM	LDA IND X	LDX IMM		LDY Z Page	LDA Z Page	LDX Z Page		TAY	LDA IMM	TAX		LDY ABS	LDA ABS	LDX ABS		A
B	BCS	LDA IND, Y			LDY Z Page X	LDA Z Page X	LDX Z Page Y		CLV	LDA ABS, Y	TSX		LDY ABS, X	LDA ABS, X	LDX ABS, Y		B
C	CPY IMM	CMP IND X			CPY Z Page	CMP Z Page	DEC Z Page		INY	CMP IMM	DEX		CPY ABS	CMP ABS	DEC ABS		C
D	BNE	CMP IND, Y				CMP Z Page X	DEC Z Page X		CLD	CMP ABS, Y				CMP ABS, X	DEC ABS, X		D
E	CPX IMM	SBC IND X			CPX Z Page	SBC Z Page	INC Z Page		INX	SBC IMM	NOP		CPX ABS	SBC ABS	INC ABS		E
F	BEQ	SBC IND Y				SBC Z Page X	INC Z Page X		SED	SBC ABS, Y				SBC ABS X	INC ABS, X		F

APPENDIX. This table to be set to cover a page. It shoud be easy to read. There will probably be no more than 1/4 page of explanatory text.

APPENDIX
MEMORY MAP

Address	
FFFF	Operating System ROM
E000	I/O ROM
D000	Not used
C000	Cartridge Slot A
A000	Cartridge Slot B
8000	Unused RAM
2800	DOS and FMS RAM — RAM available to user if system lacks diskette drive.
0700	OS RAM
0200	STACK
0100	USED BY CARTRIDGE
00FB	Unused RAM (see note)
00B0	USED BY CARTRIDGE
0000	

Note

The map shows the Page Zero RAM available to you if you have the Assembler Cartridge inserted and you are executing an Assembly Langage program. If you have the BASIC Cartridge inserted, and you are using a BASIC program to call an Assembly Language routine, then ~~the~~ only 6 bytes of Page Zero RAM are available to you, namely 00CB to 00D1.

APPENDIX

ERRORS

When an error occurs, the TV speaker is given a short "beep" and the error number is displayed.

Errors numbered less than 100 refer to the Assembler Cartridge.

ERROR NUMBER	
1.	The memory available is insufficient for the program to be assembled.
2.	For the Command "DEL xx,yy" the number xx cannot be found.
3.	There is an error in specifying an address. (Mini-assembler)
4.	The file named cannot be loaded.
5.	Undefined reference.
6.	Error in syntax of a statement.
7.	Lable defined more than once, inconsistently.
8.	Buffer overflow.
9.	There is no label before "=".
10.	The value of an expression is greater than 255.
11.	A null string has been used.
12.	The address or address type specified is incorrect.
13.	Phase error. An inconsistent result has been found from Pass 1 to Pass 2.
14.	Undefined forward reference.
15.	Line is too large.
16.	Assembler does not recognize the source statement.
17.	Line number is too large.
18.	LOMEM Command was attempted after other command(s) or instruction(s). LOMEM, if used, must be the first command.

APPENDIX

ERRORS

Errors numbered more than 100 refer to the Operating System and the Disk Operating System. For a complete list of DOS errors, refer to the DOS manual.

128	You hit the BREAK key during an I/O operation.
130	You specified a non-existent device.
132	The command is invalid for the device.
136	EOF. End of file read has been reached. This error may occur when reading from cassette.
137	A record was longer than 256 characters.
138	The device specified in the command does not respond. Make sure the device is connected to the consol and powered.
139	The device specified in the command does not return an Acknowledge signal.
140	Serial framing error.
142	Serial framing error
143	Serial data checksum error
144	Device done error
145	Diskette error--read-after-write comaprison failed.
146	
162	Exxax Disk full.
165	File name error

SPECIAL SYMBOLS (cont'd)

Register names:

A

X

Y

S

P

ASSEMBLER Mnemonics

list in colums alphabetically

MCS6501-MCS6505 MICROPROCESSOR INSTRUCTION SET – ALPHABETIC SEQUENCE

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

APPENDIX

SPECIAL SYMBOLS

Below we give a list of special symbols that have a restricted meaning to the Assembler. You should avoid using these symbols as a matter of course. Most attempts to use these symbols in any but their special sense will result in errors. They may be used, without their special meaning, in comments and in the operands of memory reservation Directives (.BYTE, .WORD).

; The semicolon is used to indicate the start of a comment. Everything between the semicolon and RETURN appears in the listed form of the program and is ignored by the Assembler. When comments take more than one line, start each new line with a semicolon.

The # sign is used as the first symbol of an immediate operand, as in LDX #24

$ The $ sign is used before numbers to signify that they are to be interpreted as hex numbers. For example, LDX #$34

* The asterisk is used to signify the value of the current location counter. For example, the ~~instruction~~ following line gives the symbol HERE a value equal to 5 more than the number in the current location counter:

20 HERE=*+5

Example:

100 ~~18~~ *=$911
110 ~~19~~ PLA
120 ~~20~~ *=*+$F
130 ~~21~~ TAX

~~In this example, assembling line 18 causes the locations counter to be $0911~~ 100

When this example is assembled, line ~~18~~ 100 causes the location counter to be $0911, PLA is placed in location $0911, line 120 causes the location counter to be increased from $0912 to $0921, and TAX is placed in $0921.

APPENDIX

Charactr set and code of
Atari Extended ASCII (8-bit code)

N.	N+ϕϕ	N+2ϕ	N+40	N+6ϕ	N+8ϕ	N+Eϕ
ϕϕ	♥	b̸	@	♦		
ϕ1	├	!	A	a		
ϕ2	▕	"	B	b		
ϕ3	┘	#	C	c		
ϕ4	┤	$	D	d		
ϕ5	┐	%	E	e		
ϕ6	╱	&	F	f		
ϕ7	╲	'	G	g		
ϕ8	◢	(	H	h		
ϕ9	▗	)	I	i		
ϕA	◤	*	J	j		
ϕB	▝	+	K	k		
ϕC	▘	,	L	l		
ϕD	▔	-	M	m		
ϕE	▁	.	N	n		
ϕF	▖	/	O	o		
1ϕ	♣	ϕ	P	p		
11	┌	1	Q	q		
12	─	2	R	r		
13	┼	3	S	s		
14	●	4	T	t		
15	▄	5	U	u		
16	▎	6	V	v		
17	┬	7	W	w		
18	┴	8	X	x		
19	▌	9	Y	y		
1A	└	:	Z	z		
1B	ESC E	;	[	♠	EOL	
1C	↑ ↑	<	\	|	DEL LINE	
1D	↓ ↓	=	]	CLEAR ↰	INS LINE	BELL
1E	← ←	>	^	BACKSP ◀	CLR TAB	DEL CHAR
1F	→ →	?	_	TAB ▶	SET TAB	INS CHAR

This table to be set to cover one page. No more than 1/4 page of text--probably none.